A Portable Kernel Abstraction For Low-Overhead Ephemeral Mapping Management

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox
Department of Computer Science
Rice University, Houston, Texas 77005, USA
{kdiaa,anupamc,alc}@cs.rice.edu

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of interprocess communication to the implementation of process tracing and debugging. With succeeding generations of processors the cost of creating ephemeral mappings is increasing, particularly when an ephemeral mapping is shared by multiple processors.

To reduce the cost of ephemeral mapping management within an operating system kernel, we introduce the `sf_buf` ephemeral mapping interface. We demonstrate how in several kernel subsystems — including pipes, memory disks, sockets, `execve()`, `ptrace()`, and the vnode pager — the current implementation can be replaced by calls to the `sf_buf` interface.

We describe the implementation of the `sf_buf` interface on the 32-bit *i386* architecture and the 64-bit *amd64* architecture. This implementation reduces the cost of ephemeral mapping management by reusing wherever possible existing virtual-to-physical address mappings. We evaluate the `sf_buf` interface for the pipe, memory disk and networking subsystems. Our results show that these subsystems perform significantly better when using the `sf_buf` interface. On a multiprocessor platform interprocessor interrupts are greatly reduced in number or eliminated altogether.

1 Introduction

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of interprocess communication to the implementation of process tracing and debugging. To create an ephemeral mapping two actions are required: the allocation of a temporary kernel virtual address and the modification of the virtual-to-physical address mapping. To date, these actions have been performed through separate interfaces. This paper demonstrates the benefits of combining these actions under a single interface.

This work is motivated by the increasing cost of ephemeral mapping creation, particularly, the increasing cost of modifications to the virtual-to-physical mapping. To see this trend, consider the latency in processor cycles for the `invlpg` instruction across several generations of the IA32 architecture. This instruction invalidates the Translation Look-aside Buffer (TLB) entry for the given virtual address. In general, the operating system must issue this instruction when it changes a virtual-to-physical mapping. When this instruction was introduced in the 486, it took 12 cycles to execute. In the Pentium, its latency increased to 25 cycles. In the Pentium III, its latency increased to $\sim$100 cycles. Finally, in the Pentium 4, its latency has reached $\sim$500 to $\sim$1000 cycles. So, despite a factor of three decrease in the cycle time between a high-end Pentium III and a high-end Pentium 4, the cost of a mapping change measured in wall clock time has actually increased.

Furthermore, on a multiprocessor, the cost of ephemeral mapping creation can be significantly higher if the mapping is shared by two or more processors. Unlike data cache coherence, TLB coherence is generally implemented in software by the operating system [2, 12]: The processor initiating a mapping change issues an interprocessor interrupt (IPI) to each of the processors that share the mapping; the interrupt handler that is executed by each of these processors includes

an instruction, such as `invlpg`, that invalidates that processor's TLB entry for the mapping's virtual address. Consequently, a mapping change is quite costly for all processors involved.

In the past, TLB coherence was only an issue for multiprocessors. Today, however, some implementations of Simultaneous Multi-Threading (SMT), such as the Pentium 4's, require the operating system to implement TLB coherence in a single-processor system.

To reduce the cost and complexity of ephemeral mapping management within an operating system kernel, we introduce the `sf_buf` ephemeral mapping interface. Like Mach's *pmap* interface [11], our objective is to provide a machine-independent interface enabling variant, machine-specific implementations. Unlike pmap, our `sf_buf` interface supports allocation of temporary kernel virtual addresses. We describe how various subsystems in the operating system kernel benefit from the `sf_buf` interface.

We present the implementation of the `sf_buf` interface on two representative architectures, *i386*, a 32-bit architecture, and *amd64*, a 64-bit architecture. This implementation is efficient: it performs creation and destruction of ephemeral mappings in O(1) expected time on *i386* and O(1) time on *amd64*. The `sf_buf` interface enables the automatic reuse of ephemeral mappings so that the high cost of mapping changes can be amortized over several uses. In addition, this implementation of the `sf_buf` interface incorporates several techniques for avoiding TLB coherence operations, eliminating the need for costly IPIs.

We have evaluated the performance of the pipe, memory disk and networking subsystems using the `sf_buf` interface. Our results show that these subsystems benefit significantly from its use. For the `bw_pipe` program from the lmbench benchmark [10] the `sf_buf` interface improves performance up to 168% on one of our test platforms. In all of our experiments the number of TLB invalidations is greatly reduced or eliminated.

The rest of the paper is organized as follows. The next two sections motivate this work from two different perspectives: First, Section 2 describes the many uses of ephemeral mappings in an operating system kernel; second, Section 3 presents the execution costs for the machine-level operations used to implement ephemeral mappings. We define the `sf_buf` interface and its implementation on two representative architectures in Section 4. Section 5 summarizes the lines of code reduction in an operating system kernel from using the `sf_buf` interface. Section 6 presents an experimental evaluation of the `sf_buf` interface. We present related work in Section 7 and conclude in Section 8.

2 Ephemeral Mapping Usage

We use FreeBSD 5.3 as an example to demonstrate the use of ephemeral mappings. FreeBSD 5.3 uses ephemeral mappings in a wide variety of places, including the implementation of pipes, memory disks, `sendfile()`, sockets, `execve()`, `ptrace()`, and the vnode pager.

2.1 Pipes

Conventional implementations of Unix pipes perform two copy operations to transfer the data from the writer to the reader. The writer copies the data from the source buffer in its user address space to a buffer in the kernel address space, and the reader later copies this data from the kernel buffer to the destination buffer in its user address space.

In the case of large data transfers that fill the pipe and block the writer, FreeBSD uses ephemeral mappings to eliminate the copy operation by the writer, reducing the number of copy operations from two to one. The writer first determines the set of physical pages underlying the source buffer, then *wires* each of these physical pages disabling their replacement or page-out, and finally passes the set to the receiver through the object implementing the pipe. Later, the reader obtains the set of physical pages from the pipe object. For each physical page, it creates an ephemeral mapping that is private to the current CPU and is not used by other CPUs. Henceforth, we refer to this kind of mapping as a CPU-private ephemeral mapping. The reader then copies the data from the kernel virtual address provided by the ephemeral mapping to the destination buffer in its user address space, destroys the ephemeral mapping, and *unwires* the physical page reenabling its replacement or page-out.

2.2 Memory Disks

Memory disks have a pool of physical pages. To read from or write to a memory disk a CPU-private ephemeral mapping for the desired pages of the memory disk is created. Then the data is copied between the ephemerally mapped pages

and the read/write buffer provided by the user. After the read or write operation completes, the ephemeral mapping is freed.

2.3 `sendfile(2)` and Sockets

The zero-copy `sendfile(2)` system call and zero-copy socket send use ephemeral mappings in a similar way. For zero-copy send the kernel wires the physical pages corresponding to the user buffer in memory and then creates ephemeral mappings for them. For `sendfile()` it does the same for the pages of the file. The ephemeral mappings persist until the corresponding `mbuf` chain is freed, e.g., when TCP acknowledgments are received. The kernel then frees the ephemeral mappings and unwires the corresponding physical pages. These ephemeral mappings are not CPU-private because they need to be shared among all the CPUs — any CPU may use the mappings to retransmit the pages.

Zero-copy socket receive uses ephemeral mappings to implement a form of *page remapping* from the kernel to the user address space [6, 4, 8]. Specifically, the kernel allocates a physical page, creates an ephemeral mapping to it, and injects the physical page and its ephemeral mapping into the network stack at the device driver. After the network interface has stored data into the physical page, the physical page and its mapping are passed upward through the network stack. Ultimately, when an application asks to receive this data, the kernel determines if the application's buffer is appropriately aligned and sized so that the kernel can avoid a copy by replacing the application's current physical page with its own. If so, the application's current physical page is freed, the kernel's physical page replaces it in the application's address space, and the ephemeral mapping is destroyed. Otherwise, the ephemeral mapping is used by the kernel to copy the data from its physical page to the application's.

2.4 `execve(2)`

The `execve(2)` system call transforms the calling process into a new process. The new process is constructed from the given file. This file is either an executable or data for an interpreter, such as a shell. If the file is an executable, FreeBSD's implementation of `execve(2)` uses the ephemeral mapping interface to access the image header describing the executable.

2.5 `ptrace(2)`

The `ptrace(2)` system call enables one process to trace or debug another process. It includes the capability to read or write the memory of the traced process. To read from or write to the traced process's memory, the kernel creates CPU-private ephemeral mappings for the desired physical pages of the traced process. The kernel then copies the data between the ephemerally mapped pages and the buffer provided by the tracing process. The kernel then frees the ephemeral mappings.

2.6 Vnode Pager

The vnode pager creates ephemeral mappings to carry out I/O. These ephemeral mappings are not CPU private. They are used for paging to and from file systems with small block sizes.

3 Cost of Ephemeral Mappings

We focus on the hardware trends that motivate the need for the `sf_buf` interface. In particular, we measure the costs of local and remote TLB invalidations in modern processors. The act of invalidating an entry from a processor's own TLB is called a local TLB invalidation. A remote TLB invalidation, also referred to as TLB shoot-down, is the act of a processor initiating invalidation of an entry from another processor's TLB. When an entry is invalidated from all TLBs in a multiprocessor environment, it is called a global TLB invalidation.

We examine two microbenchmarks: one to measure the cost of a local TLB invalidation and another to measure the cost of a remote TLB invalidation. We modify the kernel to add a custom system call that implements these microbenchmarks. For local invalidation, the system call invalidates a page mapping from the local TLB 100,000 times. For remote invalidation, IPIs are sent to invalidate the TLB entry of the remote CPUs. The remote invalidation is also repeated 100,000 times in the experiment. We perform this experiment on the Pentium Xeon processor and the Opteron processor. The Xeon is an *i386* processor while the Opteron is an *amd64* processor. The Xeon processor implements SMT and has two virtual processors. The Opteron machine has two physical processors. The Xeon operates at 2.4 GHz while the Opteron operates at 1.6 GHz.

For the Xeon the cost of a local TLB invalidation is around 500 CPU cycles when the page table entry (PTE) resides in the data cache, and about 1,000 cycles when it does not. On a Xeon machine with a single physical processor but two virtual processors, the CPU initiating a remote invalidation has to wait for about 4,000 CPU cycles until the remote TLB invalidation completes. On a Xeon machine with two physical processors and four virtual processors, that time increases to about 13,500 CPU cycles.

For the Opteron a local TLB invalidation costs around 95 CPU cycles when the PTE exists in the data cache, and 320 cycles when it does not. Remote TLB invalidations on an Opteron machine with two physical processors cost about 2,030 CPU cycles.

4 Ephemeral Mapping Management

We first present the ephemeral mapping interface. Then, we describe two distinct implementations on representative architectures, *i386* and *amd64*, emphasizing how each implementation is optimized for its underlying architecture. This section concludes with a brief characterization of the implementations on the three other architectures supported by FreeBSD 5.3.

4.1 Interface

The ephemeral mapping management interface consists of four functions that either return an ephemeral mapping object or require one as a parameter. These functions are `sf_buf_alloc()`, `sf_buf_free()`, `sf_buf_kva()`, and `sf_buf_page()`. Table 1 shows the full signature for each of these functions. The ephemeral mapping object is entirely opaque; none of its fields are public. For historical reasons, the ephemeral mapping object is called an `sf_buf`.

`sf_buf_alloc()` returns an `sf_buf` for the given physical page. A physical page is represented by an object called a `vm_page`. An implementation of `sf_buf_alloc()` may, at its discretion, return the same `sf_buf` to multiple callers if they are mapping the same physical page. In general, the advantages of shared `sf_buf`s are (1) that fewer virtual-to-physical mapping changes occur and (2) that less kernel virtual address space is used. The disadvantage is the added complexity of reference counting. The flags argument to `sf_buf_alloc()` is either 0 or one or more of the following values combined with bitwise or:

- "private" denoting that the mapping is for the private use of the calling thread;
- "no wait" denoting that `sf_buf_alloc()` must not sleep if it is unable to allocate an `sf_buf` at the present time; instead, it may return `NULL`; by default, `sf_buf_alloc()` sleeps until an `sf_buf` becomes available for allocation;
- "interruptible" denoting that the sleep by `sf_buf_alloc()` should be interruptible by a signal; if `sf_buf_alloc()`'s sleep is interrupted, it may return `NULL`.

If the "no wait" option is given, then the "interruptible" option has no effect. The "private" option enables some implementations, such as the one for *i386*, to reduce the cost of virtual-to-physical mapping changes. For example, the implementation may avoid remote TLB invalidation. Several uses of this option are described in Section 2 and evaluated in Section 6.

`sf_buf_free()` frees an `sf_buf` when its last reference is released.

`sf_buf_kva()` returns the kernel virtual address of the given `sf_buf`.

`sf_buf_page()` returns the physical page that is mapped by the given `sf_buf`.

4.2 *i386* Implementation

Conventionally, the *i386*'s 32-bit virtual address space is split into user and kernel spaces to avoid the overhead of a context switch on entry to and exit from the kernel. Commonly, the split is 3GB for the user space and 1GB for the kernel space. In the past, when physical memories were much smaller than the kernel space, a fraction of the kernel space would be dedicated to a permanent one-to-one, virtual-to-physical mapping for the machine's entire physical memory. Today, however, *i386* machines frequently have physical memories in excess of their kernel space, making such a *direct* mapping an impossibility.

To accommodate machines with physical memories in excess of their kernel space, the *i386* implementation allocates a configurable amount of the kernel space and uses it to implement a *virtual-to-physical mapping cache* that is indexed by the physical page. In other words, an access to this cache provides a physical page and receives

`struct sf_buf *`	`sf_buf_alloc(struct vm_page *`*`page`*`, int `*`flags`*`)`
`void`	`sf_buf_free(struct sf_buf *`*`mapping`*`)`
`vm_offset_t`	`sf_buf_kva(struct sf_buf *`*`mapping`*`)`
`struct vm_page *`	`sf_buf_page(struct sf_buf *`*`mapping`*`)`

Table 1: Ephemeral Mapping Interface

a kernel virtual address for accessing the provided physical page. An access is termed a cache hit if the physical page has an existing virtual-to-physical mapping in the cache. An access is termed a cache miss if the physical page does not have a mapping in the cache and one must be created.

The implementation of the mapping cache consists of two structures containing `sf_buf`s: (1) a hash table of valid `sf_buf`s that is indexed by physical page and (2) an inactive list of unused `sf_buf`s that is maintained in least-recently-used order. An `sf_buf` can appear in both structures simultaneously. In other words, an unused `sf_buf` may still represent a valid mapping.

Figure 1 defines the *i386* implementation of the `sf_buf`. It consists of six fields: an immutable virtual address, a pointer to a physical page, a reference count, a pointer used to implement a hash chain, a pointer used to implement the inactive list, and a CPU mask used for optimizing CPU-private mappings. An `sf_buf` represents a valid mapping if and only if the pointer to a physical page is valid, i.e., it is not `NULL`. An `sf_buf` is on the inactive list if and only if the reference count is zero.

The hash table and inactive list of `sf_buf`s are initialized during kernel initialization. The hash table is initially empty. The inactive list is filled as follows: A range of kernel virtual addresses is allocated by the ephemeral mapping module; for each virtual page in this range, an `sf_buf` is created, its virtual address initialized, and inserted into the inactive list.

The first action by the *i386* implementation of `sf_buf_alloc()` is to search the hash table for an `sf_buf` mapping the given physical page. If one is found, then the next two actions are determined by that `sf_buf`'s `cpumask`. First, if the executing processor does not appear in the `cpumask`, a local TLB invalidation is performed and the executing processor is added to the `cpumask`. Second, if the given flags do not include "private" and the `cpumask` does not include all processors, a remote TLB invalidation is issued to those processors missing from the `cpumask` and those processors are added to the `cpumask`. The final three actions by `sf_buf_alloc()` are (1) to remove the `sf_buf` from the inactive list if its reference count is zero, (2) to increment its reference count, and (3) to return the `sf_buf`.

If, however, an `sf_buf` mapping the given page is not found in the hash table by `sf_buf_alloc()`, the least recently used `sf_buf` is removed from the inactive list. If the inactive list is empty and the given flags include "no wait", `sf_buf_alloc()` returns `NULL`. If the inactive list is empty and the given flags do not include "no wait", `sf_buf_alloc()` sleeps until an inactive `sf_buf` becomes available. If `sf_buf_alloc()`'s sleep is interrupted because the given flags include "interruptible", `sf_buf_alloc()` returns `NULL`.

Once an inactive `sf_buf` is acquired by `sf_buf_alloc()`, it performs the following five actions. First, if the inactive `sf_buf` represents a valid mapping, specifically, if it has a valid physical page pointer, then it must be removed from the hash table. Second, the `sf_buf`'s physical page pointer is assigned the given physical page, the `sf_buf`'s reference count is set to one, and the `sf_buf` is inserted into the hash table. Third, the page table entry for the `sf_buf`'s virtual address is changed to map the given physical page. Fourth, TLB invalidations are issued and the `cpumask` is set. Both of these operations depend on the state of the old page table entry's accessed bit and the mapping options given. If the old page table entry's accessed bit was clear, then the mapping cannot possibly be cached by any TLB. In this case, no TLB invalidations are issued and the `cpumask` is set to include all processors. If, however, the old page table entry's accessed bit was set, then the mapping options determine the action taken. If the given flags include "private", then a local TLB invalidation is performed and the `cpumask` is set to contain the executing processor. Otherwise, a global TLB invalidation is performed and the `cpumask` is set

to include all processors. Finally, the sf_buf is returned.

The implementation of sf_buf_free() decrements the sf_buf's reference count, inserting the sf_buf into the free list if the reference count becomes zero. When an sf_buf is inserted into the free list, a sleeping sf_buf_alloc() is awakened.

The implementations of sf_buf_kva() and sf_buf_page() return the corresponding field from the sf_buf.

4.3 *amd64* Implementation

The *amd64* implementation of the ephemeral mapping interface is trivial because of this architecture's 64-bit virtual address space.

During kernel initialization, a permanent, one-to-one, virtual-to-physical mapping is created within the kernel's virtual address space for the machine's entire physical memory using 2MB superpages. Also, by design, the inverse of this mapping is trivially computed, using a single arithmetic operation. This mapping and its inverse are used to implement the ephemeral mapping interface. Because every physical page has a permanent kernel virtual address, there is no recurring virtual address allocation overhead associated with this implementation. Because this mapping is trivially invertible, mapping a physical page back to its kernel virtual address is easy. Because this mapping is permanent there is never a TLB invalidation.

In this implementation, the sf_buf is simply an alias for the vm_page; in other words, an sf_buf pointer references a vm_page. Consequently, the implementations of sf_buf_alloc() and sf_buf_page() are nothing more than cast operations evaluated at compile-time: sf_buf_alloc() casts the given vm_page pointer to the returned sf_buf pointer; conversely, sf_buf_page() casts the given sf_buf pointer to the returned vm_page pointer. Furthermore, none of the mapping options given by the flags passed to sf_buf_alloc() requires any action by this implementation: it never performs a remote TLB invalidation so distinct handling for "private" mappings serves no purpose; it never blocks so "interruptible" and "no wait" mappings require no action. The implementation of sf_buf_free() is the empty function. The only function to have a non-trivial implementation is sf_buf_kva(): It casts an sf_buf pointer to a vm_page pointer, dereferences that pointer to obtain the vm_page's physical address, and applies the inverse direct mapping to that physical address to obtain a kernel virtual address.

4.4 Implementations For Other Architectures

The implementations for *alpha* and *ia64* are identical to that of *amd64*. Although the *sparc64* architecture has a 64-bit virtual address space, its virtually-indexed and virtually-tagged cache for instructions and data complicates the implementation. If two or more virtual-to-physical mappings for the same physical page exist, then to maintain cache coherence either the virtual addresses must have the same *color*, meaning they conflict with each other in the cache, or else caching must be disabled for all mappings to the physical page [5]. To make the best of this, the *sparc64* implementation is, roughly speaking, a hybrid of the *i386* and *amd64* implementations: The permanent, one-to-one, virtual-to-physical mapping is used when its color is compatible with the color of the user-level address space mappings for the physical page. Otherwise, the permanent, one-to-one, virtual-to-physical mapping cannot be used, so a virtual address of a compatible color is allocated from a free list and managed through a dictionary as in the *i386* implementation.

5 Using the sf_buf Interface

Of the places where the FreeBSD kernel utilizes ephemeral mappings, only three were non-trivially affected by the conversion from the original implementation to the sf_buf-based implementation: The conversion of pipes eliminated 42 lines of code; the conversion of zero-copy receive eliminated 306 lines of code; and the conversion of the vnode pager eliminated 18 lines of code. Most of the eliminated code was for the allocation of temporary virtual addresses. For example, to minimize the overhead of allocating temporary virtual addresses, each pipe maintained its own, private cache of virtual addresses that were obtained from the kernel's general-purpose allocator.

```
struct sf_buf {
        LIST_ENTRY(sf_buf) list_entry;  /* hash list */
        TAILQ_ENTRY(sf_buf) free_entry; /* inactive list */
        struct          vm_page *m;     /* currently mapped page */
        vm_offset_t     kva;            /* virtual address of mapping */
        int             ref_count;      /* usage of this mapping */
        cpumask_t       cpumask;        /* cpus on which mapping is valid */
};
```

Figure 1: The *i386* Ephemeral Mapping Object (`sf_buf`)

6 Performance Evaluation

This section presents the experimental platforms and evaluation of the `sf_buf` interface on the pipe, memory disk and network subsystems.

6.1 Experimental Platforms

The experimental setup consisted of five platforms. The first platform is a Pentium Xeon 2.4 GHz machine, with hyper-threading enabled, having 2 GB of memory. We refer to this platform as Xeon-HTT. Due to hyper-threading the Xeon-HTT has two virtual CPUs on a single physical processor. The next three platforms are identical to Xeon-HTT but have different processor configurations. The second platform runs a uniprocessor kernel resulting in having a single virtual and physical processor. Henceforth, we refer to this platform as Xeon-UP. The third platform has two physical CPUs, each with hyper-threading disabled; we refer to this platform as Xeon-MP. The fourth platform has two physical CPUs with hyper-threading enabled, resulting in having four virtual CPUs. We refer to this platform as Xeon-MP-HTT. Unlike Xeon-UP, multiprocessor kernels run on the other Xeon platforms. The Xeon has an *i386* architecture. Our fifth platform is a dual processor Opteron model 242 (1.6 GHz) with 3 GB of memory. Henceforth, we refer to this platform as Opteron-MP. The Opteron has an *amd64* architecture. All the platforms run FreeBSD 5.3.

6.2 Executive Summary of Results

This section presents an executive summary of the experimental results. For the rest of this paper we refer to the kernel using the `sf_buf` interface as the sf_buf kernel and the kernel using the original techniques of ephemeral mapping management as the original kernel. Each experiment is performed once using the sf_buf kernel and once using the original kernel on each of the platforms. For all experiments on all platforms, the sf_buf kernel provides noticeable performance improvements.

For the Opteron-MP performance improvement is due to two factors: (1) complete elimination of virtual address allocation cost and (2) complete elimination of local and remote TLB invalidations. Under the original kernel, the machine independent code always allocates a virtual address for creating an ephemeral mapping. The corresponding machine independent code, under the sf_buf kernel, does not allocate a virtual address but makes a call to the machine dependent code. The cost of virtual address allocation is avoided in the *amd64* machine dependent implementation of the `sf_buf` interface which returns the permanent one-to-one physical-to-virtual address mappings. Secondly, since the ephemeral mappings returned by the `sf_buf` interface are permanent, all local and remote TLB invalidations for ephemeral mappings are avoided under the sf_buf kernel. The above explanation holds true for all experiments on the Opteron-MP and, hence, we do not repeat the explanation for the rest of the paper.

For the various platforms on the Xeon, the performance improvement under the sf_buf kernel were due to: (1) reduction of physical-to-virtual address allocation cost and (2) reduction of local and remote TLB invalidations. On the *i386* architecture, the `sf_buf` interface maintains a cache of physical-to-virtual address mappings. While creating an ephemeral mapping under the sf_buf kernel, a cache hit results in reuse of a physical-to-virtual mapping. The associated cost is lower than the cost of allocating a new virtual address which is done under the original kernel. Further, a cache hit avoids local and remote TLB invalidations which would have been required under the original kernel. Secondly, if an ephemeral map-

ping is declared CPU-private, it requires no remote TLB invalidations on a cache miss under the sf_buf kernel. For each of our experiments in the following sections we articulate the reasons for performance improvement under the sf_buf kernel. Unless stated otherwise, the sf_buf kernel on a Xeon machine uses a cache of 64K entries of physical-to-virtual address mappings, where each entry corresponds to a single physical page. This cache can map a maximum footprint of 256 MB. For some of the experiments we vary this cache size to study its effects.

The Xeon-UP platform outperforms all other Xeon platforms when the benchmark is single threaded. Only, the web server is multi-threaded, thus only it can exploit symmetric multi-threading (Xeon-HTT), multiple processors (Xeon-MP), or the combination of both (Xeon-MP-HTT). Moreover, Xeon-UP runs a uniprocessor kernel which is not subject to the synchronization overhead incurred by multiprocessor kernels running on the other Xeon platforms.

6.3 Pipes

This experiment used the lmbench `bw_pipe` program [10] under the sf_buf kernel and the original kernel. This benchmark creates a Unix `pipe` between two processes, transfers 50 MB through the pipe in 64 KB chunks and measures the bandwidth obtained. Figure 2 shows the result for this experiment on our test platforms. The sf_buf kernel achieved 67%, 129%, 168%, 113% and 22% higher bandwidth than the original kernel for the Xeon-UP, Xeon-HTT, Xeon-MP, Xeon-MP-HTT and the Opteron-MP respectively. For the Opteron-MP, the performance improvement is due to the reasons explained in Section 6.2. For the Xeon platforms, the small set of physical pages used by the benchmark are mapped repeatedly resulting in a near 100% cache-hit rate and complete elimination of local and remote TLB invalidations as shown in Figure 3. For all the experiments in this paper we count the number of remote TLB invalidations issued and not the number of remote TLB invalidations that actually happen on the remote processors.

6.4 Memory Disks

This section presents two experiments — one using Disk Dump (`dd`) and another using the PostMark benchmark [9] — to characterize the effect of the `sf_buf` interface on memory disks.

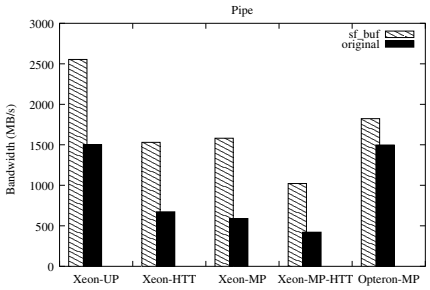

Figure 2: Pipe bandwidth in MB/s

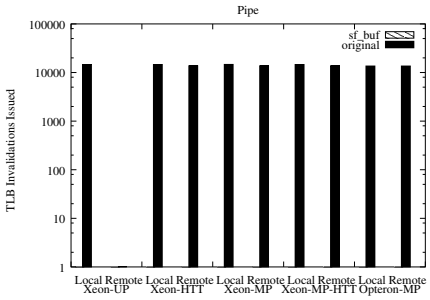

Figure 3: Local and remote TLB invalidations issued for the pipe experiment

6.4.1 Disk Dump (`dd`)

This experiment uses `dd` to transfer a memory disk to the null device using a block size of 64 KB and observes the transfer bandwidth. We perform this experiment for two sizes of memory disks — 128 MB and 512 MB. The size of the `sf_buf` cache on the Xeons is 64K entries, which can map a maximum of 256 MB, larger than the smaller memory disk but smaller than the larger one. Under the sf_buf kernel two configurations are used — one using the private mapping option and the other eliminating its use and thus creating default shared mappings.

Figures 4 and 6 show the bandwidth obtained on each of the platforms for the 128 MB disk and 512 MB disk respectively. For the Opteron-MP using the `sf_buf` interface increases the bandwidth by about 37%. On the Xeons, the `sf_buf` interface increases the bandwidth by up to 51%.

Using the private mapping option has no effect on the Opteron-MP because all local and remote TLB invalidations are avoided by the use of permanent, one-to-one physical-to-virtual mappings. Since there is no `sf_buf` cache on the Opteron-MP, similar performance is obtained on both disk sizes.

For the Xeons, the 128 MB disk can be mapped entirely by the `sf_buf` cache causing no local and remote TLB invalidations even when the private mapping option is eliminated. This is shown in Figure 5. Hence, using the private mapping option has negligible effect for the 128 MB disk as shown in Figure 4. However, the 512 MB disk cannot be mapped entirely by the `sf_buf` cache. The sequential disk access of `dd` causes almost a 100% cache-miss under the sf_buf kernel. Using the private mapping option reduces the cost of these cache misses by eliminating remote TLB invalidations and thus improves the performance, which is shown in Figure 6. As shown in Figure 7, the use of the private mapping option eliminates all remote TLB invalidations from all Xeon platforms for the 512 MB memory disk.

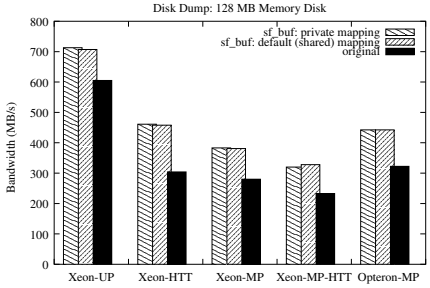

Figure 4: Disk dump bandwidth in MB/s for 128 MB memory disk

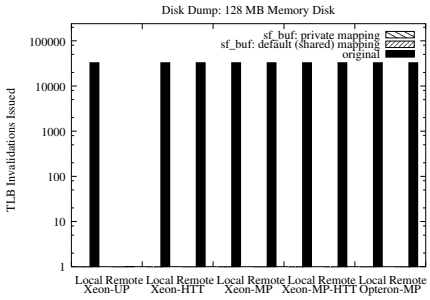

Figure 5: Local and remote TLB invalidations issued for the disk dump experiment on 128 MB memory disk

6.4.2 PostMark

PostMark is a file system benchmark simulating an electronic mail server workload [9]. It creates a pool of continuously changing files and measures the transaction rates where a transaction is creating, deleting, reading from or appending to

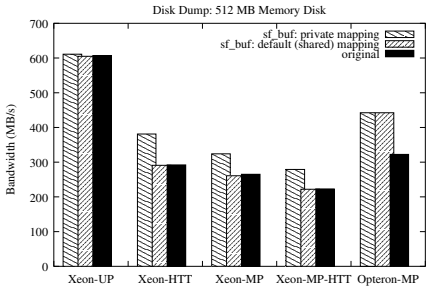

Figure 6: Disk dump bandwidth in MB/s for 512 MB memory disk

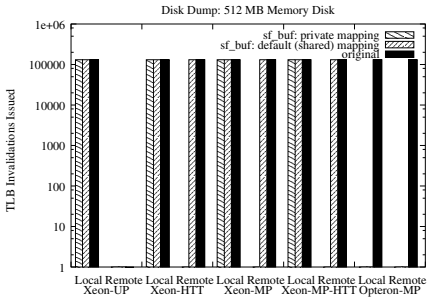

Figure 7: Local and remote TLB invalidations issued for the disk dump experiment on 512 MB memory disk

a file. We used the benchmark's default parameters, i.e., block size of 512 bytes and file sizes ranging from 500 bytes up to 9.77 KB.

We used a 512 MB memory disk for the PostMark benchmark. We used the three prescribed configurations of PostMark. The first configuration has 1,000 initial files and performs 50,000 transactions. The second has 20,000 files and performs 50,000 transactions. The third configuration has 20,000 initial files and performs 100,000 transactions.

PostMark reports the number of transactions performed per second (TPS), and it measures the read and write bandwidth obtained from the system. Figure 8 shows the TPS obtained on each of our platforms for the largest configuration of PostMark. Corresponding results for read and write bandwidths are shown in Figure 9. The results for the two other configurations of PostMark exhibit similar trends and, hence, are not shown in the paper.

For the Opteron-MP, using the `sf_buf` interface increased the TPS by about 11% to about 27%. Read and write bandwidth increased by about 11% to about 17%.

For the Xeon platforms, using the `sf_buf` in-

terface increased the TPS by about 4% to about 13%. Read and write bandwidth went up by about 4% to 15%. The maximum footprint of the PostMark benchmark is about 150 MB under the three configurations used and is completely mapped by the `sf_buf` cache on the Xeons. We did not eliminate the use of the private mapping option on the Xeons for the sf_buf kernel as there were no remote TLB invalidations under these workloads. The performance improvement on the Xeons is thus due to the elimination of local and remote TLB invalidations as shown in Figure 10.

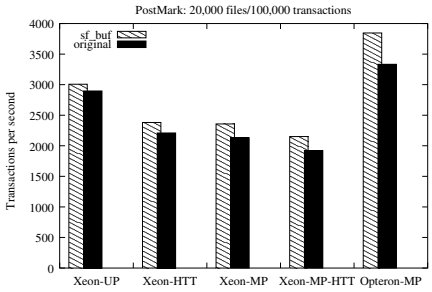

Figure 8: Transactions per second for PostMark with 20,000 files and 100,000 transactions

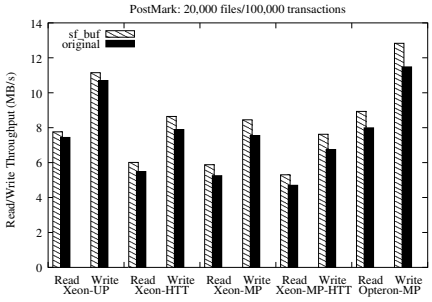

Figure 9: Read/Write Throughput (in MB/s) for PostMark with 20,000 files and 100,000 transactions

6.5 Networking Subsystem

This section uses two sets of experiments — one using `netperf` and another using a web server — to examine the effects of the `sf_buf` interface on the networking subsystem.

6.5.1 Netperf

This experiment examines the throughput achieved between a netperf client and server on the same machine. TCP socket send and receive

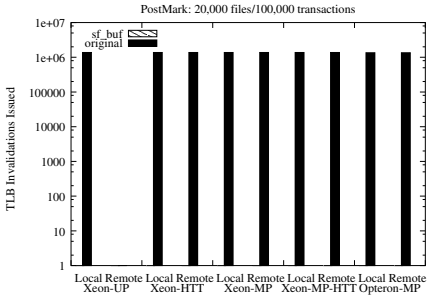

Figure 10: Local and remote TLB invalidations issued for PostMark with 20,000 files and 100,000 transactions

buffer sizes are set to 64 KB for this experiment. Sockets are configured to use zero copy send. We perform two sets of experiments on each platform, one using the default Maximum Transmission Unit (MTU) size of 1500 bytes and another using a large MTU size of 16K bytes.

Figures 11 and 12 show the network throughput obtained under the sf_buf kernel and the original kernel on each of our platforms. The larger MTU size yields higher throughput because less CPU time is spent doing TCP segmentation. The throughput improvements from the `sf_buf` interface on all platforms range from about 4% to about 34%. Using the larger MTU size makes the cost of creation of ephemeral mappings a bigger factor in network throughput. Hence, the performance improvement is higher when using the `sf_buf` interface under this scenario.

Reduction in local and remote TLB invalidations explain the above performance improvement as shown in Figures 13 and 14. The `sf_buf` interface greatly reduces TLB invalidations on the Xeons, and completely eliminates them on the Opteron-MP.

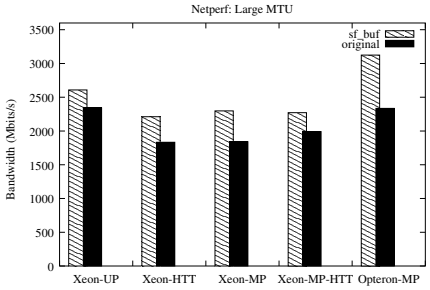

Figure 11: Netperf throughput in Mbits/s for large MTU

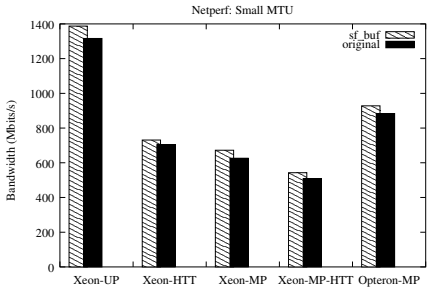

Figure 12: Netperf throughput in Mbits/s for small MTU

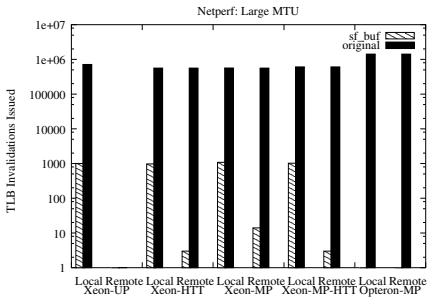

Figure 13: Local and remote TLB invalidations issued for Netperf experiments with large MTU

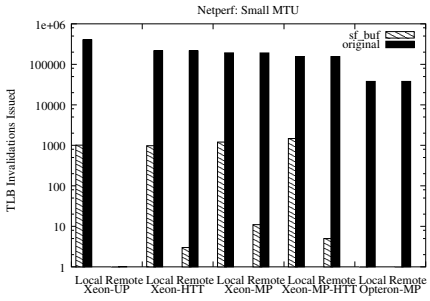

Figure 14: Local and remote TLB invalidations issued for Netperf experiments with small MTU

6.5.2 Web Server

We used apache 2.0.50 as the web server on each of our platforms. We ran an emulation of 30 concurrent clients on a separate machine to generate a workload on the server. The server and client machines were connected via a Gigabit Ethernet link. Apache was configured to use `sendfile(2)`. For this experiment we measure the throughput obtained from the server and count the number of local and remote TLB invalidations on the server machine. The web server was subject to real workloads of web traces from NASA and Rice University's Computer Science Department that have been used in published literature [7, 15]. For the rest of this paper we refer to these workloads as the NASA workload and the Rice workload respectively. These workloads have footprints of 258.7 MB and 1.1 GB respectively.

Figures 15 and 16 show the throughput for all the platforms using both the sf_buf kernel and the original kernel for the NASA and the Rice workloads respectively. For the Opteron-MP, the sf_buf kernel improves performance by about 6% for the NASA workload and about 14% for the Rice workload. The reasons behind these performance improvements are the same as described earlier in Section 6.2.

For the Xeons, using the sf_buf kernel results in performance improvement of up to about 7%. This performance improvement is a result of the reduction in local and remote TLB invalidations as shown in Figures 17 and 18.

For the above experiments the Xeon platforms employed an `sf_buf` cache of 64K entries. To study the effect of the size of this cache on web server throughput we reduced it down to 6K entries. A smaller cache causes more misses, thus increasing the number of TLB invalidations. Implementation of the `sf_buf` interface on the *i386* architecture employs an optimization which avoids TLB invalidations if the page table entry's (PTE) access bit is clear. With TCP checksum offloading enabled, the CPU does not touch the pages to be sent, and as a result the corresponding PTEs have their access bits clear and on a cache miss TLB invalidation is avoided. With TCP checksum offloading disabled, the CPU touches the pages and the corresponding PTEs, causing TLB invalidations on cache misses. So for each cache size we did two experiments, one with TCP checksum offloading enabled and the other by disabling it.

Figure 19 shows the throughput for the NASA workload on the Xeon-MP for the above experiment. For larger cache size slightly higher throughput is obtained because of more reduction in local and remote TLB invalidations as shown in Figure 20. Also, enabling checksum offloading brings local and remote TLB invalidations further down because of the access bit optimization. Reducing the cache size from 64K to 6K entries does not significantly reduce throughput because the hit rate of the ephemeral mapping cache drops from nearly 100% to about 82%. This lower cache hit rate is sufficient to avoid any noticeable performance degradation.

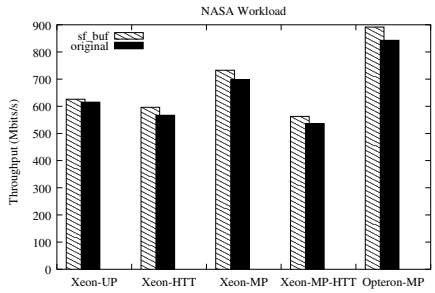

Figure 15: Throughput (in Mbits/s) for the NASA workload

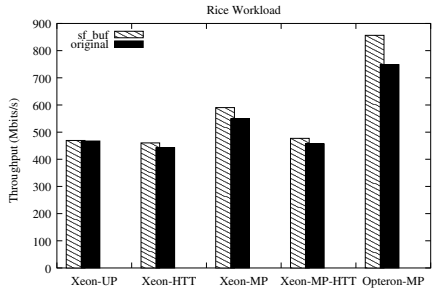

Figure 16: Throughput (in Mbits/s) for the Rice workload

7 Related Work

Chu describes a per process mapping cache for zero-copy TCP in Solaris 2.4 [6]. Since the cache is not shared among all processes in the system its benefits are limited. For example a multi-processed web server using FreeBSD's `sendfile(2)`, like apache 1.3, will not get the maximum benefit from the cache if more than one process transmit the same file. In this case the file pages are the same for all processes so having a common cache would serve best.

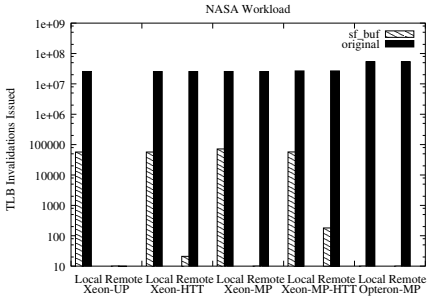

Figure 17: Local and remote TLB invalidations issued for the NASA workload

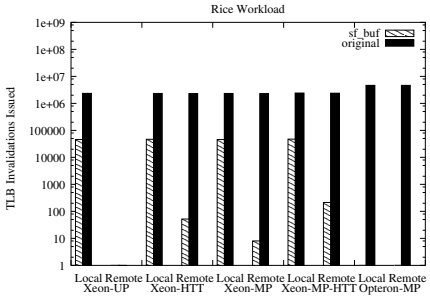

Figure 18: Local and remote TLB invalidations issued for the Rice workload

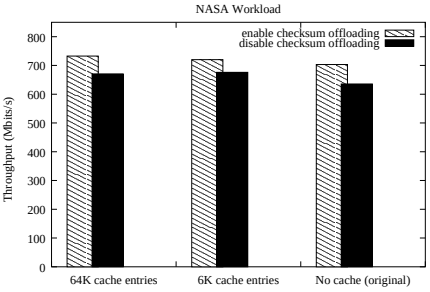

Figure 19: Throughput (in Mbits/s) for the Nasa workload on Xeon-MP with the `sf_buf` cache having 64K or 6K entries and the original kernel and with TCP checksum offloading enabled or disabled

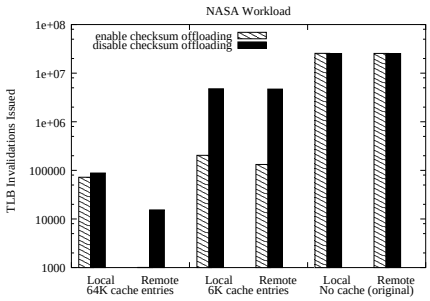

Figure 20: Local and remote TLB invalidations issued for the NASA workload on Xeon-MP with the `sf_buf` cache having 64K or 6K entries and the original kernel and with TCP checksum offloading enabled or disabled

Ruan and Pai extend the mapping cache to be shared among all processes [13]. This cache is `sendfile(2)`-specific. Our work extends the benefits of the shared mapping cache to additional kernel subsystems providing substantial modularity and performance improvements. We provide a uniform API for processor specific implementations. We also study the effect of the cache on multiprocessor systems.

Bonwick and Adams [3] describe Vmem, a generic resource allocator, where kernel virtual addresses are viewed as a type of resource. However, the goals of our work are different from that of Vmem. In the context of kernel virtual address resource, Vmem's goal is to achieve fast allocation and low fragmentation. However, it makes no guarantee that the allocated kernel virtual addresses are "safe", i.e., they require no TLB invalidations. In contrast, the `sf_buf` interface returns kernel virtual addresses that are completely safe in most cases, requiring no TLB invalidations. Additionally, the cost of using the `sf_buf` interface is small.

Bala et al. [1] design a software cache of TLB entries to provide fast access to entries on a TLB miss. This cache mitigates the costs of a TLB miss. The goal of our work is entirely different: to maintain a cache of ephemeral mappings. Because our work re-uses address mappings, it can augment such a cache of TLB entries. This is because the mappings corresponding to the physical pages with entries in the `sf_buf` interface do not need to change in the software TLB cache. In other words, the effectiveness of such a cache (of TLB entries) can be increased with the `sf_buf` interface.

Thekkath and Levy [14] explore techniques for achieving low-latency communication and implement a low-latency RPC system. They point out re-mapping as one of the sources of the cost of communication. On multiprocessors, this cost is increased due to TLB coherency operations [2]. The `sf_buf` interface obviates the need for re-mapping and hence lowers the cost of communication.

8 Conclusions

Modern operating systems create ephemeral virtual-to-physical mappings for a variety of purposes, ranging from the implementation of interprocess communication to the implementation of process tracing and debugging. The hardware costs of creating these ephemeral mappings are generally increasing with succeeding generations of processors. Moreover, if an ephemeral mapping is to be shared among multiprocessors, those processors must act to maintain the consistency of their TLBs. In this paper we have provided a software solution to alleviate this problem.

In this paper we have devised a new abstraction to be used in the operating system kernel, the ephemeral mapping interface. This interface allocates ephemeral kernel virtual addresses and virtual-to-physical address mappings. The interface is low cost, and greatly reduces the number of costly interprocessor interrupts. We call our ephemeral mapping interface as the `sf_buf` interface. We have described its implementation in the FreeBSD-5.3 kernel on two representative architectures — the *i386* and the *amd64*, and outlined its implementation for the three other architectures supported by FreeBSD. Many kernel subsystems—pipes, memory disks, sockets, `execve()`, `ptrace()`, and the vnode pager—benefit from using the `sf_buf` interface. The `sf_buf` interface also centralizes redundant code from each of these subsystems, reducing their overall size.

We have evaluated the `sf_buf` interface for the pipe, memory disk and networking subsystems. For the `bw_pipe` program of the lmbench benchmark [10] the bandwidth improved by up to about 168% on one of our platforms. For memory disks, a disk dump program resulted in about 37% to 51% improvement in bandwidth. For the PostMark benchmark [9] on a memory disk we demonstrate up to 27% increase in transaction throughput. The `sf_buf` interface increases netperf throughput by up to 34%. We also demonstrate tangible benefits for a web server workload

with the `sf_buf` interface. In all these cases, the ephemeral mapping interface greatly reduced or completely eliminated the number of TLB invalidations.

Acknowledgments

We wish to acknowledge Matthew Dillon of the DragonFly BSD Project, Tor Egge of the FreeBSD Project, and David Andersen, our shepherd. Matthew reviewed our work and incorporated it into DragonFly BSD. He also performed extensive benchmarking and developed the implementation for CPU-private mappings that is used on the *i386*. Tor reviewed parts of the *i386* implementation for FreeBSD. Last but not least, David was an enthusiastic and helpful participant in improving the presentation of our work.

References

[1] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *First Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, California, Nov. 1994.

[2] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, Dec. 1989.

[3] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference*, Boston, Massachussetts, June 2001.

[4] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Operating Systems Design and Implementation*, pages 277–291, 1996.

[5] R. Cheng. Virtual address cache in Unix. In *Proceedings of the 1987 Summer USENIX Conference*, pages 217–224, 1987.

[6] H.-K. J. Chu. Zero-copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference*, pages 253–264, Jan. 1996.

[7] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *USENIX 2004 Annual Technical Conference*, June 2004.

[8] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.

[9] J. Katcher. Postmark: A new file system benchmark. At http://www.netapp.com/tech_library/3022.html.

[10] L. McVoy and C. Stalien. Lmbench - tools for performance analysis. At http://www.bitmover.com/lmbench/, 1996.

[11] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[12] B. S. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 137–146, Litchfield Park, AZ, Dec. 1989.

[13] Y. Ruan and V. Pai. Making the Box transparent: System call performance as a first-class result. In *USENIX 2004 Annual Technical Conference*, pages 1–14, June 2004.

[14] C. A. Thekkath and H. M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[15] H. youb Kim, V. S. Pai, and S. Rixner. Increasing Web Server Throughput with Network Interface Data Caching. In *Architectural Support for Programming Languages and Operating Systems*, pages 239–250, San Jose, California, Oct. 2002.